



# Leveraging Compiler-Based Tools for Performance Portability

**Protonu Basu<sup>+</sup>**, Samuel Williams<sup>+</sup>, Brian Van Straalen<sup>+</sup>, Leonid Oliker<sup>+</sup>, Philip Colella<sup>+</sup>, Mary Hall<sup>\*</sup>

\* Lawrence Berkeley National Laboratory

\* University of Utah

### Performance and Productivity Challenge – GSRB Smooth

```
/* Helmholz */
for (k=0; k<N; k++)
    for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i] -
        temp[k][j][i];
```

```
/* Gauss-Seidel Red Black Update */
for (k=0; k<N; k++)
    for (j=0; j<N; j++)
    for (i=0; i<N; i++){
        if ((i+j+k+color)%2 == 0 )
            phi[k][j][i] = phi[k][j][i] - lambda[k][j][i] *
        (temp[k][j][i] - rhs[k][j][i]);}
```

```
const __m256d invertMask4 =
int ij,kplane=k*plane;
 int_ii_start = ii_start[planeInWavefront];
        _ij_end = ij_end[planeInMavefront];
     r(ij=_ij_start;ij<_ij_end;ij+=16){ // smooth a vector
      nt ijk=ij+kplane;
    #if defined(_PREFETCH_NEXT_PLANE_FROM_DRAM)
#warming will attempt to prefetch the next plane from DRAM one componer
double * _base = Prefetch_Pointers[prefetch_stream] + prefetch_ijk;
        _mm_prefetch((const char*)(_base+ 0),_MM_HINT_T1);
_mm_prefetch((const char*)(_base+ 8),_MM_HINT_T1);
                                                                                                                                                                                                                                               helmholtz_00 = _mm256_mul_pd(helmholtz_00,b_h2inv_splot4)
         _mm_prefetch((const char*)(_base+16),_MM_HINT_T1)
                                                                                                                                                                                                                                               helmholtz_04 = _mm256_mul_pd(helmholtz_04,b_h2inv_splat4)
          prefetch_ijk=28;if(prefetch_ijk>prefetch_ijk_end){prefetch_stream++;prefetch_ijk=prefetch_ijk_start;
                                                                                                                                                                                                                                               helmholtz_12 = _mm256_mul_pd(helmholtz_12,b_h2inv_splot4)
                                                                                                                                                                                                                                               helmholtz\_00 = \_m256\_sub\_pd(\_m256\_mul\_pd(\_m256\_mul\_pd(a\_splot4,\_m256\_lood\_pd(alpho+ijk+0)), phi\_00), helmholtz\_00) = (m256\_sub\_pd(\_m256\_mul\_pd(\_m256\_mul\_pd(a\_splot4,\_m256\_lood\_pd(alpho+ijk+0))), phi\_00), helmholtz\_00) = (m256\_sub\_pd(\_m256\_mul\_pd(a\_splot4,\_m256\_lood\_pd(alpho+ijk+0))), phi\_00), helmholtz\_00) = (m256\_sub\_pd(a\_splot4,\_m256\_lood\_pd(alpho+ijk+0))), helmholtz\_00), helmholtz\_00) = (m256\_sub\_pd(a\_splot4,\_m256\_lood\_pd(alpho+ijk+0))), helmholtz\_00) = (m256\_sub\_pd(a_splot4,\_m256\_lood\_pd(alpho+ijk+0))), helmholtz_00) = (m256\_sub\_pd(a_splot4,\_m256\_lood\_pd(alpho+ijk+0))), helmholtz_00) = (m256\_sub\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256\_lood\_pd(a_splot4,\_m256
                                                                                                                                                                                                                                               helinholtz_04 = _m256_sub_pd(_m256_mul_pd(_m256_mul_pd(_a_splot4,_m256_lcod_pd(alpho+ijk+4)),phi_04),helinholtz_04);
                 m256d helmholtz 12:
                                                                                                                                                                                                                                              helmholtz 88 = .m256 sub pdf m256 mul pdf m256 mul pdfa solat4, m256 load pdfalpho+irk+ 81),phi 881,helmholtz 880;
                                                                                                                                                                                                                                             helmholtz_12 = _m256_sub_pd(_m256_mul_pd(_m256_mul_pd(a_solat4__m256_load_pd(alpho+irk+12)),phi_12),helmholtz_12);
                                           temp_00 = _mm_load_pd(phi+ijk+ -2);
                                                                                                                                                                                                                                    _m256d nev_c00 = _m256_mul_pd(_m256_lood_pd(lombds+ijk+0),_m256_sub_pd(helmholtz_00,_m256_lood_pd(rhs+ijk+0)));
                                           temp_02 = _mm_load_pd(phi+ijk+ 0);
                                                                                                                                                                                                                                                     new_04 = _m256_mul_pd(_mu256_load_pd(lambda+ijk+ 4),_m256_sub_pd(helmholtz_04,_m256_load_pd(rhs+ijk+ 4)))
                                          temp_01 = _mm_shuffle_pd(temp_00,temp_02,1);
temp_04 = _mm_load_pd(phi+ijk+ 2);
                                                                                                                                                                                                                                                       nex_60 = _m256_mul_pd(_m256_load_pd(lambds+ijk+ 8),_m256_sub_pd(helmholtz_60,_m256_load_pd(rhs+ijk+ 8)))
                                           temp_06 = _mm_load_pd(phi+ijk+ 4);
                                                                                                                                                                                                                                                       new_12 = _m256_mul_pd(_mx256_load_pd(lambds+ijk+12),_mx256_sub_pd(helmboltz_12,_mx256_load_pd(rhs+ijk+12)))
                                           temp 03 = mm shuffle pd/temp 02.temp 04.1):
                                                                                                                                                                                                                                                        nev_60 = _m256_sub_pd(phi_60,nev_60)
                                           temp_05 = _mm_shuffle_pd(temp_04, temp_06, 1);

phi_00 = _mm256_insertf128_pd(_mm256_castpd128_pd256(temp_02), temp_04, 1);
                                                                                                                                                                                                                                                        nev 64 = .mr256 sub pd/ohi 64.nev 64)
                                            phi m1 = mm256 insertf128 pdf mm256 costpd128 pd256(temp 01).temp 03.1);
                                                                                                                                                                                                                                                       nev_68 = _m256_sub_pd(phi_66_nev_68):
    const __m256d
const __m128d
                                           phi_01 = _mm256_insertf128_pd(_mm256_castpd128_pd256(temp_03),temp_05,1);
temp_08 = _mm_load_pd(phi+ijk+ 6);
                                                                                                                                                                                                                                                       nev 17 = mr256 sub nd/rshi 17.nev 121:
                                                                                                                                                                                                                            const m256d RedBlack 80 = m056 yar ad/invertilask4, mm256 casts i256 ad/ mm256 load s i256/ ( m256i4)(RedBlackHask+ii+ 8) 111)
                                           temp 10 = mm load pd(phi+iik+ 8):
                                                                                                                                                                                                                            const _m256d RedBlock_04 = _m256_nor_pd(invertHosk4__m256_costsi256_pd(_m256_lood_si256( (_m256i*)(RedBlockHosk+ij+ 4) )))
                                           temp_07 = _mm_shuffle_pd(temp_06,temp_08,1);
temp_09 = _mm_shuffle_pd(temp_08,temp_10,1);
                                                                                                                                                                                                                            const _m256d RedBlock_00 = _m256_xor_pd(invertHosk4,_m256_costs:256_pd(_m256_load_s:256( (__m256i*)(RedBlockNosk+ij+ 8) )));
    const __m256d
                                           phi_84 = _mm256_insertf128_pd(_mm256_castpd128_pd256(temp_86),temp_88_1);
                                                                                                                                                                                                                            const __m256d ledBlock_12 = _mm256_nor_pd(invertblosk4,_mm256_costs1256_pd(_mm256_load_s1256( (__m2561*)(ledBlockblosk+ij+12) )))
   const __m256d
const __m256d
const __m128d
                                           phi_03 = _mm256_insertf128_pd(_mm256_castpd128_pd256(temp_05),temp_07,1);
phi_05 = _mm256_insertf128_pd(_mm256_castpd128_pd256(temp_07),temp_09,1);
                                                                                                                                                                                                                                                       new 80 = _mm256_blendv_pdfphi_80.new_80_RedBlack_80);
                                                                                                                                                                                                                                                       new_84 = _mr256_blendv_pd(phi_84,new_84,Red3lack_84);
                                           temp_12 = _mm_load_pd(phi+ijk+ 10);
   const __m128d
const __m128d
const __m128d
                                           temp 14 = mm load pd(phi+iik+ 12)
                                                                                                                                                                                                                                                       new_88 = _mr256_blendv_pd(phi_88,new_88,Red3lack_88);
                                          temp_11 = _mm_shuffle_pd(temp_10,temp_12,1);
temp_13 = _mm_shuffle_pd(temp_12,temp_14,1);
                                                                                                                                                                                                                                                       new_12 = _mm256_blendv_pd(phi_12,new_12,RadBlock_12);
                                                                                                                                                                                                                                                                         _mm256_store_pd(phi+ijk+ 0,nem_00);
   const __m256d
const __m256d
const __m256d
                                           phi_08 = _mm256_insertf128_pd(_mm256_castpd128_pd256(temp_10),temp_12,1);
                                           phi_07 = _mm256_insertf128_pd(_mm256_castpd128_pd256(temp_09),temp_11,1);
phi_09 = _mm256_insertf128_pd(_mm256_castpd128_pd256(temp_11),temp_13,1);
                                                                                                                                                                                                                                                                         m256 store pdfphi+irk+ 4.nex 84):
                                                                                                                                                                                                                                                                         mm256 store pd/phi+iik+ 1.new 88)
   const __m128d
const __m128d
const __m128d
                                           temp_16 = _mm_load_pd(phi+ijk+ 14)
                                                                                                                                                                                                                                                                         m256 store pd/phi+irk+12.new 12);
                                          temp_18 = _mm_load_pd(phi+ijk+ 16);
temp_15 = _mm_shuffle_pd(temp_14,temp_16,1);
temp_17 = _mm_shuffle_pd(temp_16,temp_18,1);
```

Code A: miniGMG baseline smooth operator approximately 13 lines of code

**Code B:** miniGMG optimized smooth operator approximately 170 lines of code

#### **GPU code for GSRB Smooth**

```
double * beta_k = gpu_subdomains[box].tevels[level].grids[_beta_k] + (plane+pencil+1);
double * lambda = gpu_subdomains[box].tevels[level].grids[_lembda] + (plane+pencil+1);
double hZinv = 1.8/(h*h);
int box = blockIdx.z; // CUDA 4 !!!
int subdomain_dim = gpu_subdomains[box].levels[level].dim.i;
int pencil = gpu_subdomains[box].levels[level].pencil;
int plane = gpu_subdomains[box].levels[level].plane;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              /}
/double new_phi = phi_ijk - lambda[ijk]*(helmholtz_ijk - rhs[ijk]);
                                                                                                                                                                                                                                                                                                                                                       //if(withinBounds && ((RedBlockUpdate^k)&&xl)){
// phi(ijk) = phi_ijk - lambda[ijk]*(helmholtz_ijk - rhs[ijk)); // 6588
        7/ GSRS
double nem.phi = phi_ijk - lambdo[ijk]*(helmholtz_ijk - rhs[ijk]);
phi[ijk] = (RedSlockUpdate) ? nem.phi : phi_ijk;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     int Redblocklindere u (inthousee) & Rel:
                                                                                                                                                                                                                                                                                                                                                   for(k-8;k-subdomoin_dim;k++){
  int lik = k*plone + 1*pencil + 1;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            //}
//double new.phi = phi_ijk - loebdo[ijk]*(helmholtz_ijk - rhs[ijk]);
//_f(faithindounds){
//_phi[ijk] = ((RedBlockSpdote*k)Mbcl) ? new.phi : phi_ijk;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             (7)
(adž) arkijedos aužadž) arkijedos ad 1 :
```

Code C: miniGMG optimized smooth operator for GPU, 308 lines of code for just kernel



#### **Background: Challenges**

- Performance portability
   Across fundamentally different CPU and GPU architectures
- Programmer productivity
   High performance implementations will require low-level specification in standard MPI+OpenMP, CUDA
  - Software maintainability and portability
     May require maintaining multiple implementation of same computation



# Possible ways to address the challenges

- Follow MPI and OpenMP standards
   Same code unlikely to perform well across CPU and GPU

   Low level specification may be required for high-performing OpenMP
   Vendor C and Fortran compilers not optimized for HPC workloads
  - Some domain-specific framework strategies
     Libraries, C++ template expansion, standalone DSL
     Not composable with other optimizations



#### **Compiler Based Approach**

# Autotuner

#### **Code Variants**

CHiLL is a source-to-source compiler framework with a script interface

Novel Domain-Specific optimization implemented in CHiLL

- Exploit existing compiler transformations to accomplish optimization goals
- Develop new domain-specific transformations and required analysis and code generation support
  - Supports autotuning



#### **Compiler Based Approach**

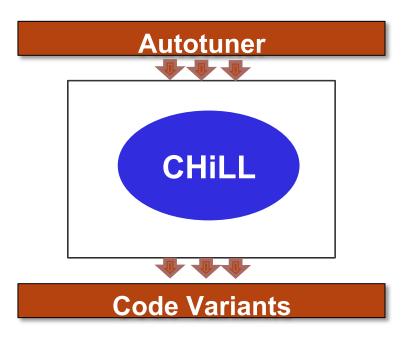
Composable transformation and code generation

Leverage rich set of existing transformations and code generation capability

Mathematically represented using polyhedral framework

 Extensible to new domainspecific transformations and decision algorithms

Compose with existing transformations





**Experience with CHILL** 

	Input	Existing Transformations	Domain-specific transformations	Autotuning
Geometric Multigrid	Sequential C computation (w/ MPI and OpenMP harness)	Communication- avoiding: fusion, tile, wavefront (skew&permute), OpenMP, CUDA	Ghost zones, Partial sums	Ghost zone depth, threading, strategy at each level of V-cycle
Tensor Contraction	Mathematical Formula	Tile, permute, scalar replacement, unroll, CUDA	Rewriting, Decision algorithm	Loop order, CUDA threading
Sparse Matrix Computation	Sequential C with CSR matrix	Tile, permute, skew, unroll, reduction, scalar expansion, OpenMP, CUDA	Generate inspectors, coalesce, makedense, compact, split, level sets	Threading, matrix repr.



#### **Performance Bottlenecks**

Stenc	Coefficie	Iteratio	Flop	Byte	Al	
7-point	Constant	Jacobi	8	24	0.33	

# Performance Limited by Memory Bandwidth!



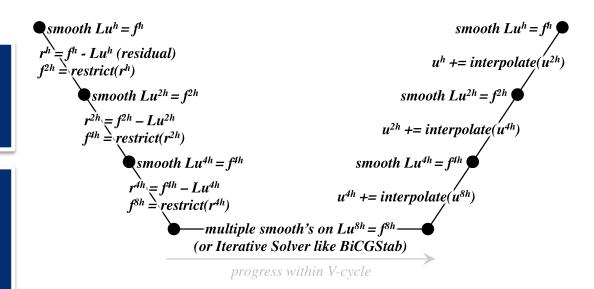
Stencil	Coefficient	Iteration	Flop	Byte	Al	beta []	
7-point	Variable	GSRB	17	80	0.21	Øěta_i[]	_
7-point	Constant	Jacobi	8	24	0.33		Increasing
13-point	Constant	Jacobi	15	24	0.63		g Flops/Byte
27-point	Constant	Jacobi	32	24	1.33		3yte
125-point	Constant	Jacobi	134	24	5.58		

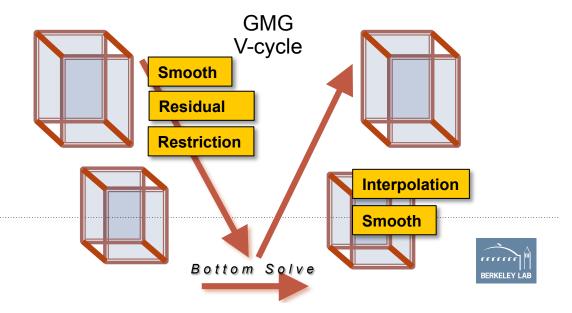


#### **Geometric Multigrid (GMG)**

MG is a hierarchical approach to solving the linear system Ax=B

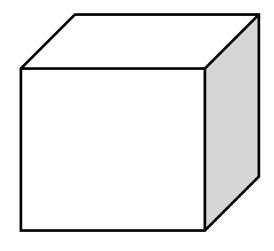
GMG solves the linear system Ax=B, where A is a stencil applied on a grid



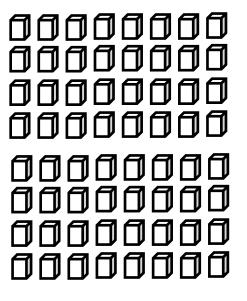


#### **Domain 256^3**

#### miniGMG



List of 64<sup>^3</sup> Boxes
Computed In Parallel (OMP)

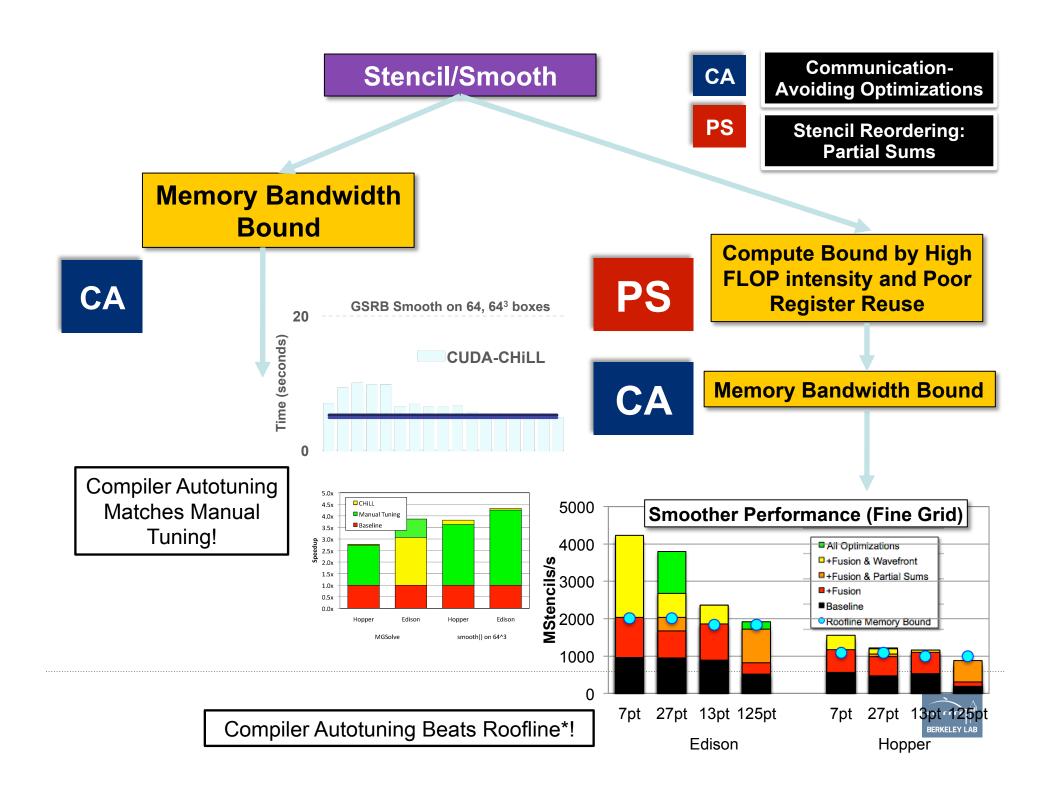


4 iterations of smooth **GMG** V-cycle 48 iterations of Smooth

Domain decomposed to MPI processes (2)

**Smooth Dominates Runtime** 



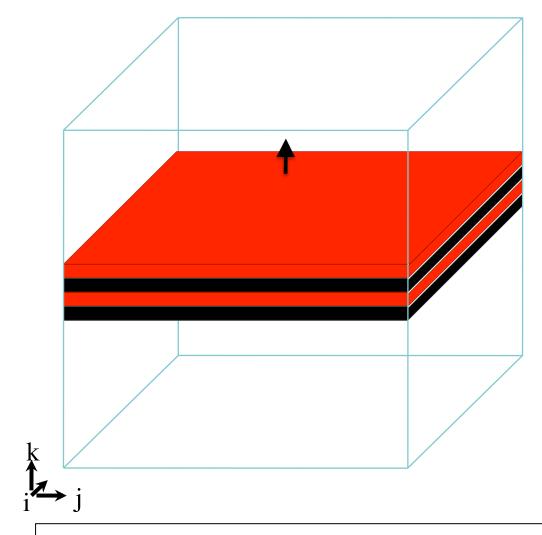


#### **Baseline GSRB Smooth**

```
S0
         for (k=0; k<N; k++)
            for (j=0; j<N; j++)
              for (i=0; i<N; i++)
                     /* statement S0 */
                     temp[k][j][i] = b * h2inv * (
 7point VC
                      beta_i[k][j][i+1] * ( phi[k][j][i+1] - phi[k][j][i] )
                      -beta_i[k][j][i] * ( phi[k][j][i] – phi[k][j][i-1] )
   stencil
                     +beta_j[k][j+1][i] * ( phi[k][j+1][i] - phi[k][j][i] )
                     -beta_j[k][j][i] * ( phi[k][j][i] - phi[k][j-1][i] )
                     +beta_k[k+1][j][i] * ( phi[k+1][j][i] - phi[k][j][i] )
                     -beta_k[k][j][i] * ( phi[k][j][i] - phi[k-1][j][i] ) );
                                                                                  S1
         for (k=0; k<N; k++)
            for (j=0; j<N; j++)
              for (i=0; i<N; i++)
GSRB
                     /* statement S1 */
                     temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i] - temp[k][j][i];
update
                                                                                             S2
         for (k=0; k<N; k++)
            for (j=0; j<N; j++)
              for (i=0; i<N; i++){
                     if ((i+j+k+color)\%2 == 0)
                     /* statement S2 */
                     phi[k][j][i] = phi[k][j][i] - lambda[k][j][i] *(temp[k][j][i] - rhs[k][j][i]);}
```



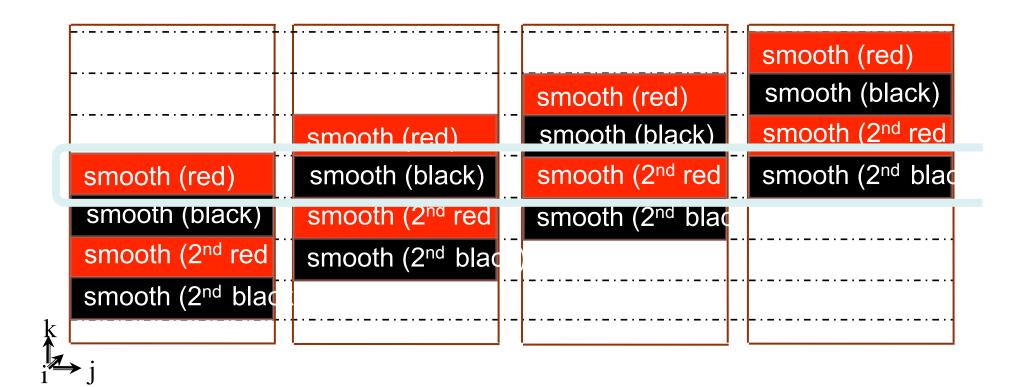
#### **Wavefront: Reducing Vertical Communication**



Wavefront fuses multiple grid sweeps reducing DRAM traffic



#### **Wavefront: Reducing Vertical Communication**



Wavefront = Loop Skew + Loop Permute

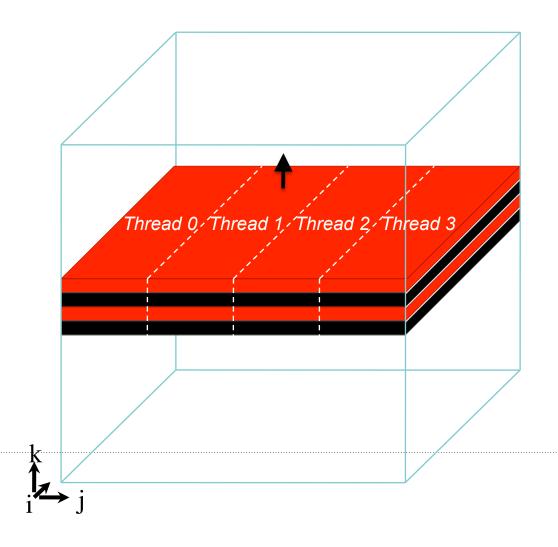
We tune to find the ghost zone depth and wavefront depth!



#### **OpenMP Code Generation: Nested Parallelism**

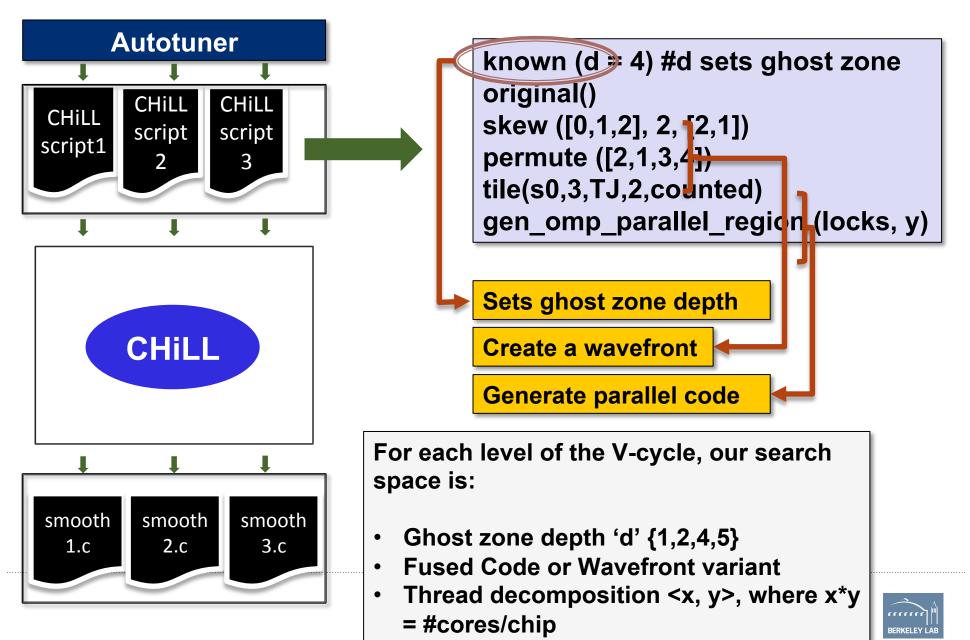
Wavefront has a larger working set

Thread blocking to manage working set

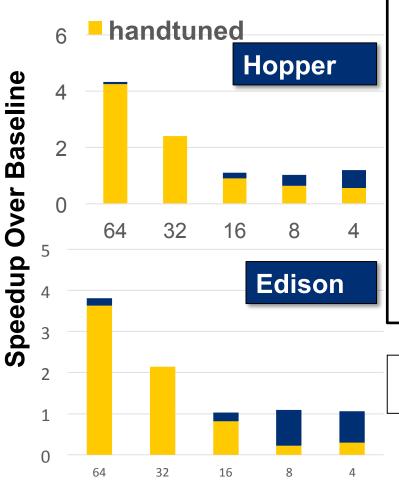




#### **Experimental Methodology**



#### Performance of GSRB Smooth



	_	nost one	Thread Decomposition <outer. inner=""></outer.>			Code Variant
Box	Edi	/Нор	Edison			Edi/Hop
64	4		<4,3>	<2,3>		Wave
32	4		<4,3>	<2,3>		Wave
16	2		<12,1>	<6,1>		Wave
8	2		<12,1>	<6,1>		Fused
4	2		<12,1>	<6,1>		Fused

Manual tuning spent considerable effort on finer 64<sup>3</sup> boxes but did not specialize for smaller boxes

Autotuning picked nested- parallelism for finer boxes; manually tuned code used intra-box threading

**Box Size** 



#### **CUDA-CHILL**

Lua/Python Interface

**CUDA-CHILL** 

CHILL

CUDA-CHILL is a thin layer built on top of CHILL to generate CUDA code

Deconstructs (tiles) a loop nest, and assigns loops to threads and blocks



#### **Parallelization via Loop Tiling**

#### **Input GSRB smooth**

```
for(box=0; box<64; box++){
  for(k=1; k<=64; k++){
    for(j=1; j<=64; j++){
     for(i=1; i<=64; i++){
      if(( i+ j + k + (color) ) % 2 == 1 ) {
          S0();
          S1();
          S2();}}}
```

```
BZ is fixed to 64 (number of boxes)
```

Tune to find best value of TX, TY (dimensions of 2D block)

BX=64(box size)/TX, BY=64(box size)/TY

#### Tiled loop nest with loops marked for blocks/threads

```
mark as block dim z (BZ=64) for(box = 0; box <= 63; box++) { for(k = 1; k <= 64; k++) { for(jj = 0; jj <= 3; jj++) { for(ii = 0; ii <= 1; ii++) { for(i = intMod(-j-k-color-1,2); i <= 31; i += 2) { S0(); S1(); } }} mark as block dim x (BX=2)
```

mark as thread dim x (TX=32)



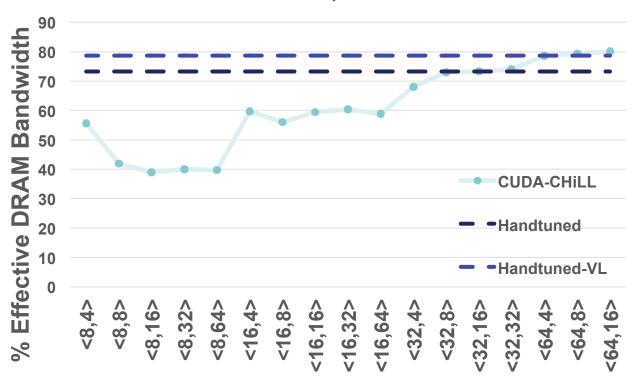
#### **CUDA-CHILL**

```
/* gsrb.lua, variable coefficient GSRB, 64<sup>3</sup> box size */
init("gsrb_mod.cu", "gsrb",0,0)
dofile("cudaize.lua") # custom commands in lua
# set up parallel decomposition, adjust via
autotuning
TI=32
TJ=4
TK=64
TZ=64
tile_by_index(0, {"box","k","j", "i"},{TZ,TK, TJ, TI},
{I1_control="bb", I2_control="kk", I3_control="jj",
I4_control="ii"},{"bb","box","kk","k","jj","j","ii","i"})
cudaize(0, "kernel GPU",
{_temp=N*N*N*N,_beta_i=N*N*N*N,
_phi=N*N*N*N},{block={"ii","jj","box"},
thread={"i","j"}},{})
```



#### Performance on K20c

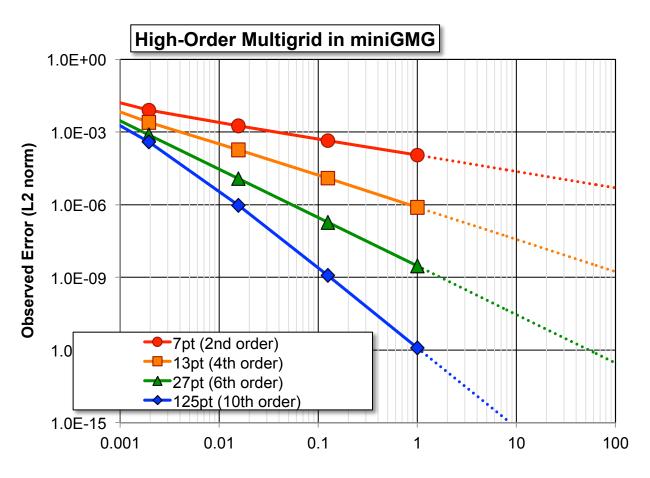
% DRAM Bandwidth achieved by GSRB Smooth on 64, 64<sup>3</sup> boxes



2D Thread Blocks <TX,TY>



#### **Higher-Order Stencils**



Requisite Memory per Vector (GB)

Higher-order stencil promise huge reduction in data movement, but maybe bottlenecked by floating-point pressure and poor register reuse



#### **Higher-Order Stencils**

Stenci	Coefficien	Iteration	Flop	Byte	Al
7-point	Constant	Jacobi	8	24	0.33
13-point	Constant	Jacobi	15	24	0.63
27-point	Constant	Jacobi	32	24	1.33
125-point	Constant	Jacobi	134	24	5.58

Higher-order stencil promise huge reduction in data movement, but maybe bottlenecked by floating-point pressure and poor register reuse



#### **Partial Sums**

#### 2D 9-point CC stencil



```
for (j=0; j<N; j++)

for (i=0; i<N; i++){

out[k][j][i] = w1*

(in[j-1][i] + in[j+1][i]

+in[j][i-1] + in[j][i+1])

+ w2 *(in[j-1][i-1] + in[j+1][i-1]

+in[j-1][i+1] + in[j+1][i+1])

+ w3* in[j][i];

}
```

Right (leading) edge of points from the input grid is reused in the next two iterations of the inner-loop

The right edge acts as the center and left edge for the next iterations

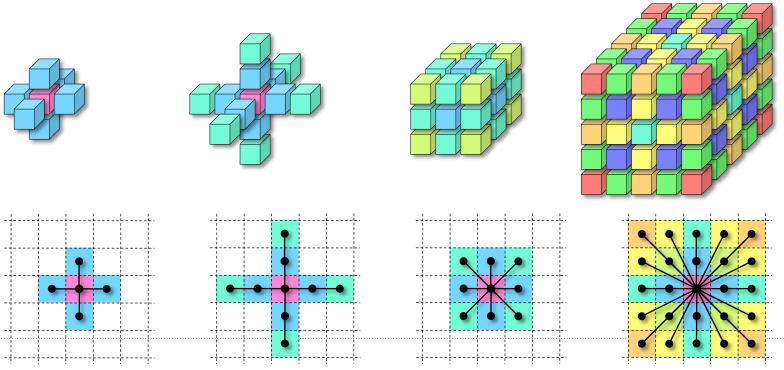


#### **Partial Sums**

For 3D stencils we pick the leading plane

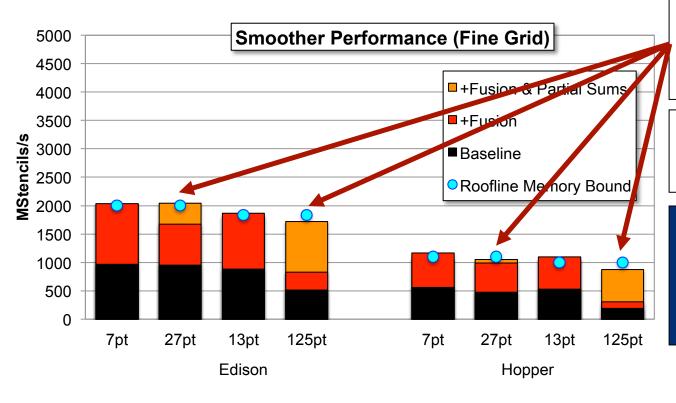
**Exploiting symmetry reduces flops significantly for 27, 125-pt stencils** 

For 125-pt stencil, 124 adds went down to 38 adds (over 3x reduction)





#### **Smooth Performance**



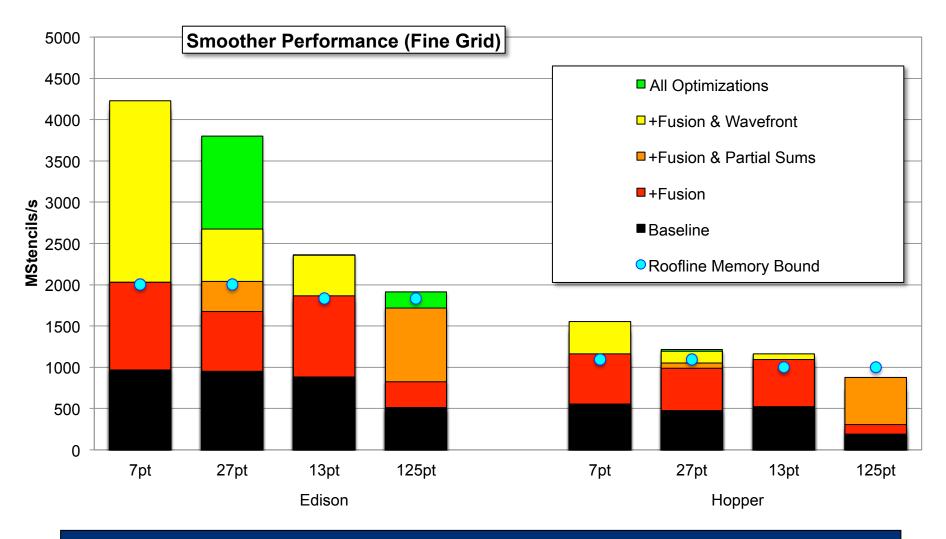
Partial sums takes the performance of the smoother to near the roofline bound

Naively one may conclude reaching the bound is the upper end of performance

Achieving memory bound implies we can now apply communication-avoiding optimizations!



#### **Smooth Performance**



**Transformation must work with other CA optimizations!** 



#### Partial Sums – CHiLL Script

```
/* jacobi_box_4_64.py, 27-pt stencil, 643 box size */
from chill import *
#select which computation to optimize
source('jacobi box 4 64.c')
procedure('smooth box 4 64')
loop(0)
original() # fuse wherever possible
#create a parallel wavefront
skew([0,1,2,3,4,5],2,[2,1])
permute([2,1,3,4])
#partial sum for high order stencils and fuse result
distribute([0,1,2,3,4,5],2)
stencil temp(0)
stencil temp(5)
fuse([2,3,4,5,6,7,8,9],1)
fuse([2,3,4,5,6,7,8,9],2)
fuse([2,3,4,5,6,7,8,9],3)
fuse([2,3,4,5,6,7,8,9],4)
```



#### **Summary and Conclusions**

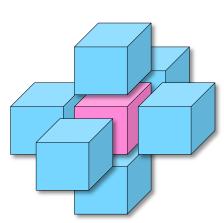
- Compiler technology can be leveraged for automated architecture-specific optimization from high-level specification for several motifs
- Compiler technology allows composing a sequence of transformations, and mixing known and novel domainspecific optimizations
  - Performance rivaling manually-tuned code and sometimes better



#### **Extra Slides**



#### **Arithmetic Intensity (AI) of Stencil Computation**



```
for (k=0; k<N; k++)
for (j=0; j<N; j++)
for (i=0; i<N; i++){

Read N³ Grid (phi_in)

Write Allocate N³ Grid (phi_out)

phi_out[k][j][i] = w1 * phi_in[k][j][i]

+ w2 * (phi_in[k+1][j][i] + phi_in[k-1][j][i]

+ phi_in[k][j+1][i] + phi_in[k][j-1][i]

+ phi_in[k][j+1][i] + phi_in[k][j-1][i]);

}

Write N³ Grid (phi_out)
```

#### Ideal cache behavior, compulsory (cold) misses only

Floating Point Ops (flops) = 
$$\frac{8 * N^3}{}$$
 = 0.33  
Data Moved (Bytes) =  $3 * N^3 * 8$ 



#### Machine Balance (Edison)



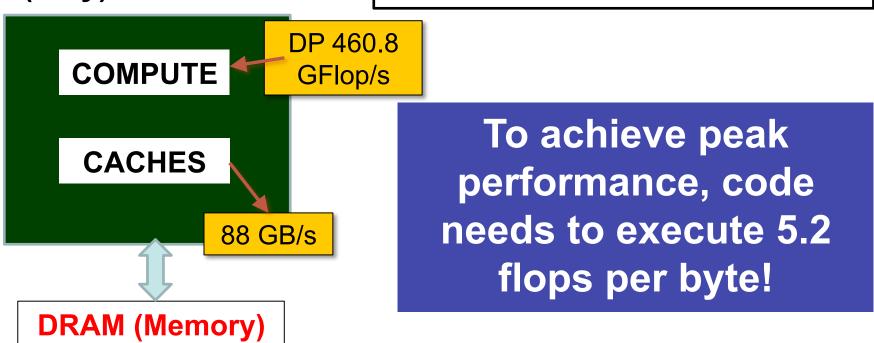
(very) Abstract Node

Floating Point Ops per second 460.8

DRAM Memory Bandwidth

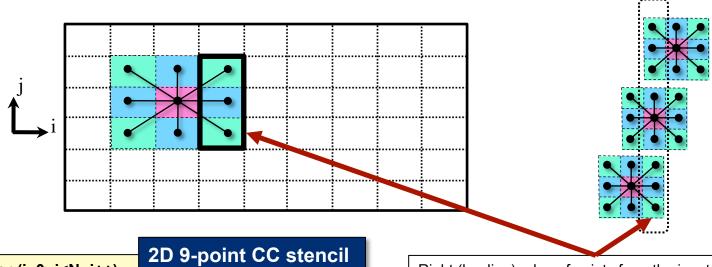
88

Machine Balance 5.2





## Opportunity for Data and Computation Reuse



```
for (j=0; j<N; j++)

for (i=0; i<N; i++){

out[k][j][i] = w1*

(in[j-1][i] + in[j+1][i]

+in[j][i-1] + in[j][i+1])

+ w2 *(in[j-1][i-1] + in[j+1][i-1]

+in[j-1][i+1] + in[j+1][i+1])

+ w3* in[j][i];

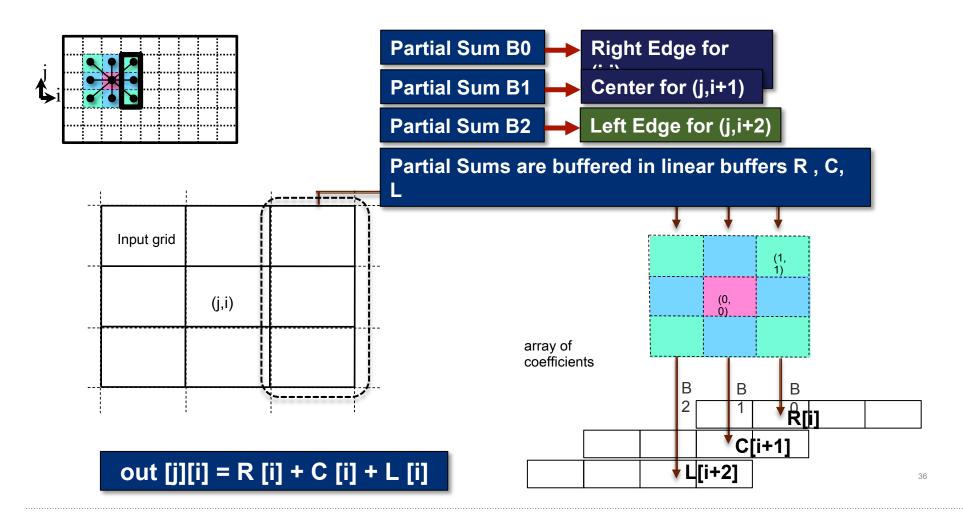
}
```

Right (leading) edge of points from the input grid is reused in the next two iterations of the inner-loop

The right edge acts as the center and left edge for the next iterations

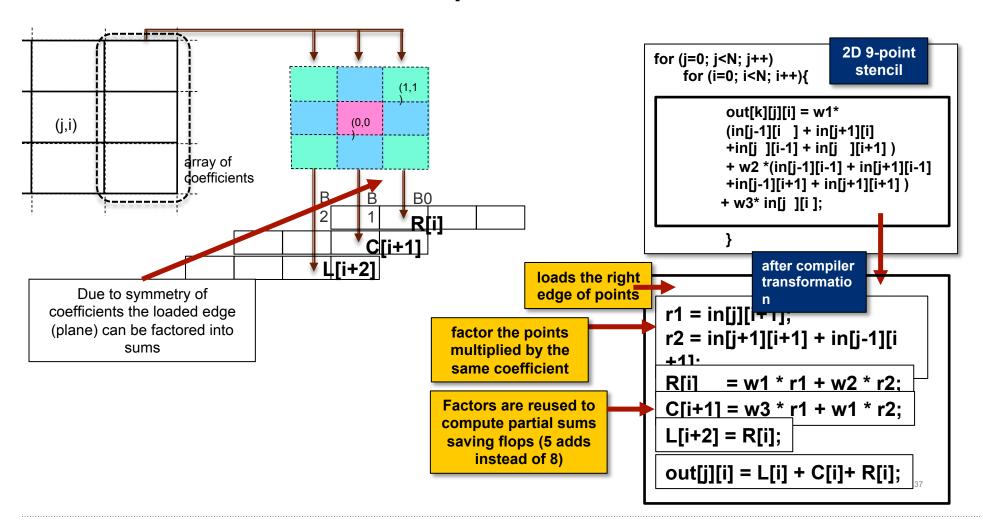


#### Buffering Partial Sums: Exploiting Reuse





# Exploiting Symmetry to Reduce Computation



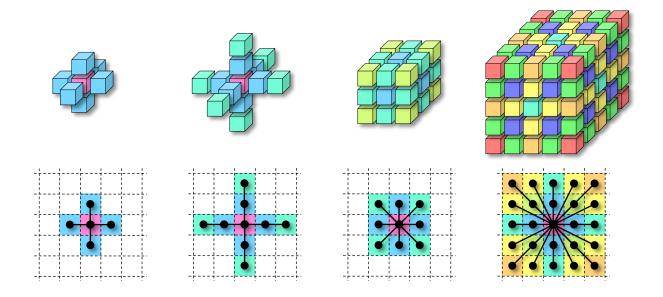


# Exploiting Symmetry to Reduce Computation

For 3D stencils we pick the leading plane

Exploiting symmetry reduces flops significantly for 27, 125-pt stencils

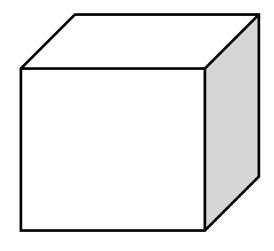
For 125-pt stencil, 124 adds went down to 38 adds (over 3x reduction)



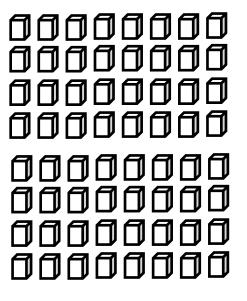


#### **Domain 256^3**

#### miniGMG



List of 64<sup>^3</sup> Boxes
Computed In Parallel (OMP)



4 iterations of smooth **GMG** V-cycle 48 iterations of Smooth

Domain decomposed to MPI processes (2)

**Smooth Dominates Runtime** 

